Parallel pipelined histogram architectures

Article

Accepted Version

Parallel pipelined histogram architectures

J. Cadenas, R. S. Sherratt and P. Huerta

Proposed here is a unique cell histogram architecture which will process *k* data items in parallel to compute 2^q histogram bins per time step. An array of $m/2^q$ cells computes an *m*-bin histogram with a speedup factor of *k*; $k \geq 2$ makes it faster than current dual-ported memory implementations. Furthermore, simple mechanisms for conflict-free storing of the histogram bins into external memory array are discussed.

Introduction: The real-time computation of a histogram is a common operation, especially in computer vision and image processing such as tone reproduction and contrast enhancement in image-processing engines of still digital cameras [1]. Parallel histogram methods do exist that exploit software and hardware techniques [2]. The two most conventional techniques for hardware-based histogram implementation use either an array of counters or a memory array. However, an implementation with an array of counters suffers from inefficient use of resources [3], therefore, most techniques use memory arrays. In general, the main challenge for parallel histogram computation using a memory array is in handling updates to a particular bin count when at least two data items map to the same bin resulting in a memory write conflict. Most mechanisms for parallel histogram computation therefore require multi-port memory arrays, where memory write conflicts have to be dealt with. Due to practical limitations, a dual-port memory array is the common solution, but then the maximum speedup is limited, up to a factor of two. This Letter argues that these challenges are easily overcome by revisiting the original idea of an array of counters, but to distribute the counting of bins in a fully pipelined manner.

Performance speedups, up to a factor of k, are achieved with the cell architecture proposed here.

Firstly, a pipelined histogram of m bins is developed using m pipeline cells. The histogram on an input data set of n items is completed in n+m-1 steps when processing one data item per time step, where m-1 is the pipeline latency. Secondly, this Letter develops pipelined arrays to process k data items per time step that complete a histogram in $n/k + \lfloor (m-1)/2^q \rfloor$ time steps, by computing 2^q bins per cell as particularly will be illustrated for k = 2 and 2^q = 2.

A pipelined histogram array: A histogram for a data set of non-negative integers where the data values correspond directly to histogram bins (such as pixels in a grey-scale image) is described by the following pseudo-code:

for i=1,…, n **do**

histogram[*data*[i]] = histogram[*data*[i]] + 1;

Assume item *data*[i] takes any value in the set of positive integers 0, …, m-1. Let r_j, s_j, j = 0, …, m-1 be a set of values such that $s_j < s_{j+1}$. Values s_j remain constant across the histogram computation while r_j represents the histogram bins. Initially, all r_j = 0. The histogram is then defined by an array of cells with each cell j computing:

if (*data*[i] == s_j) $r_j = r_j + 1$;

else $r_j = r_j$;

This definition is a degenerative case of a scalar quantiser array encoder for a uniform quantisation process with boundary distance d = 1 [4]. Unrolling the above computation over j produces a linear array of m cells with connections between cells left to right. Each cell is shown in Fig. 1. At step i, input *xin* = $data_i$, *sin* = 0 is entered in the first cell of the array. Data moves through the cells incrementing r_j for the cell where data matches *sin*; *sin* is updated for the next cell test. Each cell has a cost of a comparison and an increment. For an array the cost is essentially m comparators

and m adders. Clearly, this array allows up to m increments to operate simultaneously; a key difference to the traditional approach. The last data item enters the array after n time steps and the histogram computation time is n+m-1 steps, where a latency of m-1 steps is the cost for a data item to travel through the array. Although the occupancy of cells has been improved, this architecture still performs serial computation.

Cells for reducing array latency: Figure 2 shows a new cell architecture merging two consecutive pipeline cells into a single new cell, hence reducing the array length from m to $m/2$. In this case, each cell will generate histogram counts for $2j$ and $2j$+1, j = 0, …, $\lfloor (m-1)/2 \rfloor$; where either one bin (not both) may increment per time step. For a data item width of p bits, a comparison between the p-1 MSB bits of a data item and the *sin* value is performed. When the comparison matches, bit 0 (LSB) of the data item acts as a select signal for a data multiplexer (selecting either r^0out or r^1out in figure 2) and also as a capture enable signal for the bin count registers. It is clear that the scheme can be tailored to compute 2^q bins into a single cell. The comparison is then performed on the p-q MSB bits of a data item, while q bits act as data selectors. A q-to-2^q decoder will be required to account for register enable signals. This scheme will process a histogram of n data items in $n + \lfloor (m-1)/2^q \rfloor$ time steps, hence reducing latency of the array by a factor of 2^q. Merging 2^q pipelined cells into a single cell will account for hardware savings and consequently this shallower array of $m/2^q$ cells saves hardware compared to the previous pipelined array of m cells.

A k-way parallel pipelined histogram: Figure 3 shows cell architecture that processes two data items per time step, labeled x^0in and x^1in. Two parallel comparisons are performed between the p-1 MSB bits of both data items and the *sin* value, resulting in c_0, c_1 bits. Either, or both, bin counters per cell, r^0out and r^1out, will be updated

depending on the internal computed c_0, c_1 bit values. No count update is performed when c_0, c_1 are both zero. The specific bin count that should be updated depends on the LSB values of the two input data items to be used (the task of the logic block in figure 3.) For example, if both c_0 and c_1 are true, it may be the case that a bin count must be updated by two instead of one; this is the case when both input data items are actually of the same value. The actual value that must be updated for a bin is computed in variable *v*. Signals e_0, e_1 are register enable signals that determine an update for bin count r^0out and r^1out respectively. Figure 3 also shows the Boolean logic equations for *v*, e_0 and e_1.

In general, for *n* data items, the histogram will be computed in $n/k + \lfloor (m-1)/2^q \rfloor$ time steps with figure 3 showing an instance for $k = 2$, $2^q = 2$. A generic array is composed of $m/2^q$ cells where each cell computes 2^q bin counts, (namely, $2^q i$, for $i = 0, \ldots, 2^q-1$) in each of $j = 0,\ldots,\lfloor (m-1)/2^q \rfloor$ cells. A cell can be easily modified to process *k* data items per time step while computing 2^q bins to build arrays that can deliver a speedup factor of *k*, while reducing pipeline latency by a factor of 2^q.

Reading the histogram: After all data items have been processed by an array, the final histogram bin counts are stored across all cells in the array. It is of practical importance to have mechanisms to read the bins out from the array. One of such mechanisms is shown at the bottom dashed box of figure 3. For the $2^q = 2$ case, following the last pair of data items having been put through the pipeline array, signal *bri* is set high to force the output *Hout* to be routed either from inputs r^1out or *Hin*. In turn, *Hin* signal is connected to the *Hout* output from the neighbouring cell to the right. Input *Hin* on the rightmost cell in the array is set to zero. Histogram bin counts start to emerge from the bins computed on the leftmost cell in the array. After $m/2^{q-1}$ time steps, all histogram bins have been read out at the rate of one bin count every time step; latency was hidden with the histogram computation for the last pair of input

data item. This is convenient for storing the histogram in an external single port memory array. Similar arrangements are possible for different 2^q cases.

Results: Table 1 presents computation time and latency for the three architectures, expressed as time steps; the simple array of counters [3] and the parallel histogram with dual-ported memories [2] are included for comparison. Notice latency corresponds to the number of cells in the array. The k-way parallel architecture has an efficiency of $k2^q/m$ [5]. The efficiency of the proposed array, compared to an array of counters, is then improved by a factor of $k2^q$. The pipeline cell is efficient, since the k-speedup factor of the whole array is indeed due to the cell's internal parallelism, with a cell efficiency of $k/2^q$. In fact, a cell of a k-way parallel architecture becomes cost-optimal when compared to a cell of the (serial) pipeline architecture for the case $k = 2^q$. In this case, the pT_P cost [5] of the k-way parallel cell is $2^q(n/k) = n$ or $O(n)$, while the pT_P of a (serial) pipeline cell is also $O(n)$ as seen in Table 1.

Conclusion: A unique pipelined array of cells for the computation of a histogram with a speedup factor of k is presented. This cell architecture is able to process k data items per time step while computing 2^q histogram bins; k and q are design parameters. Significantly, no memory arrays are used for parallel histogram computation, thus no write address conflicts to such memory arrays can occur, yet the histogram bins can still be easily stored in external memory arrays. It is the careful choice of k that offers a better speedup factor than the factor of two obtained from common histogram solutions based on dual-port memories.

References

1 WEN-CHUNG, K.: 'High dynamic range imaging by fusing multiple raw images and tone reproduction', *IEEE Trans. Consum. Electron*., 2008, 54, (1), pp. 10-15

2 SHAHBAHRAMI, A., HUR, J. Y., JUULINK, B., and WONG, S.: 'FPGA implementation of parallel histogram computation', 2nd HiPEAC Workshop on Reconf. Computing, 2008, pp. 63-72

3 MULLER, S.: 'A new programmable VLSI architecture for histogram and statistics computation in different windows', Proc. Int. Conf. on Image Processing, 1995, pp. 73-76

4 MEGSON, G. M., and DIEMOZ, E.: 'Scalar quantization using a fast systolic array', *Electron. Lett.*, 1997, 33, (17), pp. 1435-1437

5 GRAMA, A., GUPTA, A., KARYPIS, G., and KUMAR, V.: 'Introduction to Parallel Computing' (Addison-Wesley, 2003)

Authors' affiliations:

J. Cadenas and R. S. Sherratt (School of Systems Engineering, University of Reading, Reading, RG6 6AY, United Kingdom)

P. Huerta (Escuela Técnica Superior, Universidad Rey Juan Carlos, Madrid, Spain)

E-mail: o.cadenas@reading.ac.uk

Figure Captions:

Fig. 1 Left: cell architecture and Right: cell computation for a pipelined histogram

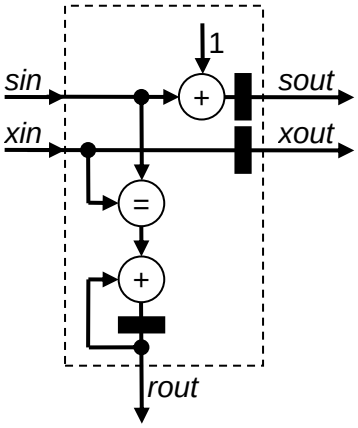
Fig. 2 Left: cell architecture and Right: a cell computation to process two bins per cell in a pipelined histogram. Latency of the array is reduced by 2.

Fig. 3 Left: cell architecture and Right: cell computation for a 2-way parallel pipelined histogram.

Table Captions:

Table 1: Parameterised performance of the presented architectures against previous implementations [2], [3].

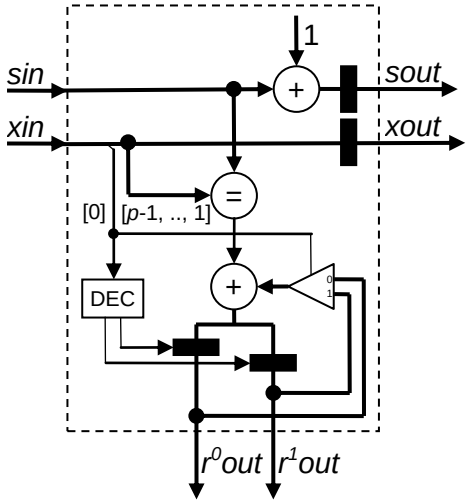
Figure 1



sout=*sin*+1;
xout=*xin*;
if(*xin*==*sin*)
  rout=*rout*+1;
else
  rout=*rout*;

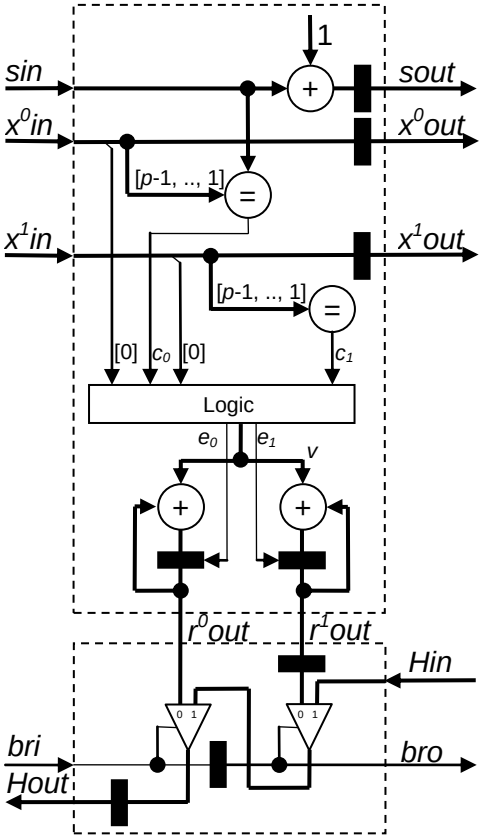
register

Figure 2



sout=*sin*+1;
xout=*xin*;
if(*xin*[*p*-1, ..., 1]==*sin*)
  switch(*xin*[0])
    case 0: $r^0out=r^0out+1$;
    case 1: $r^1out=r^1out+1$;
else
  $r^0out=r^0out$;
  $r^1out=r^1out$;

Figure 3



sout=*sin*+1;
x^0*out*=x^0*in*;
x^1*out*=x^1*in*;
c_0=(x^0*in*==*sin*);
c_1=(x^1*in*==*sin*);
$a = x^0in[0]$;
$b = x^1in[0]$;
switch({c_0,c_1})
case {0,0}: *v*=0; $e_0 = 0$; $e_1 = 0$;
case {0,1}: *v*=1; $e_0 = \bar{b}$; $e_1 = b$;
case {1,0}: *v*=1; $e_0 = \bar{a}$; $e_1 = a$;
case {1,1}: *v*=1+*a* xnor *b*;
$e_0 = \overline{a \, and \, b}$; $e_1 = a \, or \, b$;
if(e_0==1) r^0*out*=r^0*out*+*v*;
if(e_1==1) r^1*out*=r^1*out*+*v*;

Table 1

Histogram architecture	Computation time	Latency	Speedup
Parallel histogram [2]	$n/2+m/2$	$m/2$	2
Array of counters [3]	n	0	1
Pipelined array	$n+m-1$	$m-1$	1
Low latency array	$n + \lfloor (m-1)/2^q \rfloor$	$\lfloor (m-1)/2^q \rfloor$	1
k-way parallel array	$n/k + \lfloor (m-1)/2^q \rfloor$	$\lfloor (m-1)/2^q \rfloor$	k